C++学习笔记10

2022年3月19日 20:57

- 构造类操作
  - 拷贝控制操作有哪些
    - 拷贝构造函数
    - 拷贝赋值运算符
    - 移动构造函数
    - 移动赋值运算符
    - 析构函数
  - 拷贝构造函数
    - 如果任何一个构造函数的第一个参数是自身类类型的引用，且任何额外参数都有默认值，则此构造函数是拷贝构造函数

```
class Foo {
public:
    Foo();              // 默认构造函数
    Foo(const Foo&);  // 拷贝构造函数
    // ...
};
```

    - 与合成默认构造函数不同，即使我们定义了其他构造函数，编译器也会为我们合成一个拷贝构造函数
    - 对类类型成员，会使用其拷贝构造函数来拷贝；内置类型成员则直接拷贝
    - 拷贝初始化和直接初始化之间的区别
      - 当使用直接初始化的时候，我们实际上要求编译器使用普通的函数匹配来选择与我们提供的参数最匹配的构造函数
      - 当使用拷贝初始化的时候，我们要求编译器将右侧运算对象拷贝到正在创建的对象中，如果需要的话还要进行类型转换
    - 拷贝初始化发生的时间
      - 将一个对象作为实参传递给一个非引用类型的形参
      - 从一个返回类型为非引用类型的函数返回一个对象
      - 用花括号列表初始化一个数组中的元素或一个聚合类中的成员
      - 某些类类型还会对他们所分配的对象使用拷贝初始化
        - 对容器使用insert/push（拷贝初始化）
        - 用emplace的push和insert直接初始化）
      - 参数和返回值
        - 在函数调用过程中，具有非引用类的参数要进行拷贝初始化
        - 类似的，当一个函数具有非引用类型的返回类型时，返回值会用来初始化调用方的结果
    - 注意

        - 拷贝构造函数不应该是expliclt的
        - 我们可以通过控制拷贝构造函数，让类的行为发生改变
            - 类的行为像一个值
                - 当类的的值像一个值的时候，意味着它也有自己的状态，
                - 副本和原对象是完全独立的。改变副本不会对原对象有任何影响
            - 类的行为像一个指针
                - 行为像指针的类则共享状态。
                - 当我们拷贝一个这种类的对象时，副本和原对象使用相同的底层数据。改变副本会改变对象，反之亦然
- 拷贝赋值运算符
    - 与类控制其对象如何初始化一样，类也可以控制其对象如何赋值
    - 与拷贝构造函数一样，如果类未定义自己的拷贝赋值运算符。编译器会为他合成一个。
    - 重载赋值运算符
        - 重载运算符的本质上是函数，其名字由operator关键字后接标识要定义的运算符的符号组成。因此赋值运算符就是一个名为operator=的函数
        - 类似于任何其他函数，运算符函数也有一个返回类型和一个参数列表
        - 重载运算符的参数标识运算符的运算对象
            - 某些运算符必须定义为成员函数
                - 如果一个运算符是成员函数
                    - 其左侧的对象绑定到隐式的this参数中
                    - 其右侧的运算对象作为显示参数传递
        - 拷贝运算符接受一个与其所在类型相同的参数
            - 操作
            - 为了与内置类型的赋值保持一致，赋值运算符通常返回一个指向其左侧运算对象的引用
            - 标准库通常要求保存在容器中的类型要有赋值运算符，且其返回的类型是左侧运算对象的引用
    - 合成拷贝赋值与运算符
        - 对于某些类，合成拷贝赋值运算符用来禁止该类型对象的赋值。
        - 如果拷贝赋值运算符并非出于此目的，它会将右侧运算对象的每个非static成员
- 析构函数
    - 析构函数执行与构造函数相反的操作：释放对象使用的资源，并销毁对象的非static成员
    - 析构函数是类的一个成员函数，名字由波浪号接类名组成，他没有返回值也不接受参数

```
class Foo {
public:
    ~Foo(); // 析构函数
```

- 因为析构函数不接收参数，所以他不能被重载，对于一个给定的类，只有唯一一个析构函数
- 在一个析构函数中，首先执行函数体，然后销毁成员，成员按照初始化的顺序逆序销毁
- 在对象最后一次使用后，析构函数可执行类设计者希望执行的让你和守卫操作，通常为释放对象在生存期分配的所有资源
- 析构部分是隐式的。成员销毁时发生什么完全依赖与成员类型
  - 注意：隐式销毁一个内置指针类型不会销毁他所指的对象！
- 当指向一个对象或者引用离开作用域时，析构函数不会执行销毁指针指向对象的操作，而尝试销毁指针自己，对象还是没有被销毁
- 调用析构函数的条件
  - 变量离开作用域
  - 当一个对象被销毁其成员也被销毁
  - 容器（标准库或者数组）成员被销毁
  - 动态分配的对象当对他的指针delete时候
  - 临时对象：创建完表达式被销毁

○ 类值拷贝复制运算符

- 组合了析构函数和构造函数的操作
- 类似析构函数，赋值操作会销毁左边的运算对象的资源。
- 类似拷贝函数，赋值操作会从右侧运算对象拷贝数据
- 注意：赋值运算符
  - 如果将一个对象赋予它自身，赋值运算符必须能正常工作
  - 大多数赋值运算符组合了析构函数和拷贝构造函数的工作
  - 编写赋值运算符的时候，要先将右侧运算对象拷贝到一个局部临时变量中。拷贝完成后销毁左侧运算对象

○ 定义删除的函数

- 在新标准情况下，我们可以通过将拷贝构造函数和拷贝赋值运算符定义为删除的函数来阻止拷贝
- 我们定义了删除函数，但是不能使用它们
- 操作
  - 我们在参数列表后面加上=delete来指出我们希望将它定义为删除的，同时也可以通知编译器（以及读者）不希望定义这些成员

```
struct NoCopy {
    NoCopy() = default;       // 使用合成的默认构造函数
    NoCopy(const NoCopy&) = delete;               // 阻止拷贝
    NoCopy &operator=(const NoCopy&) = delete;    // 阻止赋值

    ~NoCopy() = default;      // 使用合成的析构函数
```

```
    ~NoCopy() = default;       // 使用合成的析构函数
    // 其他成员
};
```

- 对于这些类来说编译器会默认为他们合成的成员定义为删除的函数
- 如果一个类有数据成员不能默认构造，拷贝，复制，销毁，则对应的成员函数将被定义为删除的
- 对于一个具有引用成员或无法默认构造的const成员的类，编译器不会为其合成默认构造函数
- 注意
  - 析构函数是不能删除的成员

- 函数的交换操作
  - 对于哪些与重排元素一起使用的类，定义swap是非常重要的。这类算法会在需要交换两个元素时会调用swap
  - 对于标准库的函数会通常进行值的交换，而有时候我们需要定义交换指针的swap
  - 因此，假定作用域中有using声明，如果存在特殊类型的swap版本，swap调用会与之匹配，如果不存在类型的特定声明，则会使用std中的版本
  - 与拷贝控制成员不同，swap并不是必要的，但是对于分配了资源的类，定义swap可能是一种重要的优化手段
  - 小技巧：在赋值运算符中使用swap
    - 在做到将右边的值赋予左边的同时销毁了原先左边所有的对象

```
{
    using std::swap;
    swap(lhs.h, rhs.h); // 使用 HasPtr 版本的 swap
    // 交换类型 Foo 的其他成员
}

// 注意 rhs 是按值传递的，意味着 HasPtr 的拷贝构造函数
// 将右侧运算对象中的 string 拷贝到 rhs
HasPtr& HasPtr::operator=(HasPtr rhs)
{
    // 交换左侧运算对象和局部变量 rhs 的内容
    swap(*this, rhs);     // rhs 现在指向本对象曾经使用的内存
    return *this;         // rhs 被销毁，从而 delete 了 rhs 中的指针
}
```

- 注意
  - C++语言并不要求我们定义这些所有操作：可以只定义其中1-2个，而不必定义所有，但是这些操作通常应该被看作一个整体，通常只定义了一个操作而不需要定义所有操作的情况是少见的
  - 如果这个类需要一个析构函数，那么一定有必要它需要一个拷贝构造函数和一个拷贝赋值运算符
  - 我们只能对默认版本的成员函数使用=default（即默认构造函数或拷贝控制函数）
  - 大多数类应该定义默认构造函数，拷贝构造函数和拷贝赋值运算符，无论是显式还是隐式
  - 对于析构函数已删除的类型，不能定义该类型变量或释放指向该类型的动态分配对象的指针

- 移动类操作

- 移动构造函数和std：：move
  - 使用移动构造函数
    - 我们需要了解移动构造函数通常是将资源从给定对象移动而不是拷贝正在创建的对象，而且移动后的对象依然保持一个有效可析构的状态
      - 相当于是赋值一个指针
    - 能否进行移动的对象
      - 能进行移动的对象
        - 标准库容器，string，share-ptr
      - 不能进行移动的对象
        - IO,unique-ptr
  - 创建移动构造函数
    - 操作

```
StrVec::StrVec(StrVec &&s) noexcept // 移动操作不应抛出任何异常
  // 成员初始化器接管 s 中的资源
  : elements(s.elements), first_free(s.first_free), cap(s.cap)
{
    // 令 s 进入这样的状态——对其运行析构函数是安全的
    s.elements = s.first_free = s.cap = nullptr;
}
```

    - 对于一个类，一旦资源完成移动，源对象必须不再指向被移动的资源——这些资源的所有权以及归属于新创建的对象
    - noexcept：通知标准库此构造函数不会抛出任何异常，移动构造函数必须标记为noexpect
    -
- 右值引用
  - 左值与右值
    - 对于大多数来说：左值可以出现在“=”左边的东西（如i，j，k变量），右值：只能出现在“=”右边的东西（如42，36，12*5）

```
int i = 42;
int &r = i;              // 正确：r 引用 i
int &&rr = i;            // 错误：不能将一个右值引用绑定到一个左值上
int &r2 = i * 42;        // 错误：i*42 是一个右值
const int &r3 = i * 42;  // 正确：我们可以将一个 const 的引用绑定到一个右值上
int &&rr2 = i * 42;      // 正确：将 rr2 绑定到乘法结果上
```

    - 返回左值引用的函数，连同赋值，下标，解引用和前置递增/递减运算符，都是返回左值的表达式的例子。我们可以将一个左值引用绑定到这类表达式的结果上
    - 返回非引用类型的函数，连同算数，关系，位以及后置递增/递减运算符，都生成右值。我们不能将一个左值绑定到这类表达式上，但是我们可以将一个const左值引用或者一个右值引用绑定到这类表达式上
    - 变量是左值
      - 因为变量是左值，我们不能将一个右值引用绑定到一个右值引用的类型上
  - 特性
    - 只能绑定到临时对象
    - 所引用的对象将要被销毁

        - 该对象没有其他用户
        - 使用右值引用可以自由的接管所引用对象的资源
    - std：：move
        - 定义在头文件utlity中
        - move与函数
            - 如果要使用移动构造函数，必须使用move
            - 调用move的时候，需要直接调用std::move而不是move
        - std：：move与右值引用
            - 我们可以将左值转化为右值
              int &&i=222；int &&p=std::move(i);
            - 使用也代表承诺:除了赋值和销毁，我们不再使用它
    - 移动迭代器
        - 新的标准库定义了一种移动迭代器，移动迭代器解引用运算符生成一个右值引用
        - 我们可以调用make move iterator来将一个普通迭代器转换为移动迭代器
        - 可以将移动迭代器传递给uninitialized-copy
    - 注意
        - 对于一个以及被使用的右值引用，只有两种命运：赋值或者销毁，无论是内置变量还是类
            - 满足赋值需要调用“构造”函数
            - 满足销毁需要调用“析构”函数
        - 不要随意使用移动操作
            - 由于一个移后源对象的不确定状态，不要对其调用std：：move
        - 在移动构造函数和移动赋值运算符这些类实现代码之外的地方，只有当你确信需要进行移动操作且移动操作是安全的，才可以使用std：：move
- 移动赋值运算符
    - 对于这种情况，a=b的时候要先看是不是a本来就等于b，移动完成最后b将会销毁

```
StrVec &StrVec::operator=(StrVec &&rhs) noexcept
{
    // 直接检测自赋值
    if (this != &rhs) {
        free();                    // 释放已有元素
        elements = rhs.elements; // 从 rhs 接管资源
        first_free = rhs.first_free;


        cap = rhs.cap;
        // 将 rhs 置于可析构状态
        rhs.elements = rhs.first_free = rhs.cap = nullptr;
    }
    return *this;
}
```

    - noexcept：通知标准库此构造函数不会抛出任何异常，移动赋值运算符必须标记为noexpect
- 合成的移动操作
    - 默认合成移动操作的时机

        - 默认不会定义
            - 一个类定义了自己的拷贝构造函数/拷贝赋值运算符/析构函数
        - 默认会定义
            - 只有当一个类没有定义任何自己版本的拷贝控制成员，且它的所有数据成员都能移动构造或者移动赋值时，编译器才会为它合成移动构造函数或者移动赋值运算符
    - 移动操作与删除函数
        - 与拷贝操作不同，移动操作永远不会隐式定义为删除的函数
        - 如果我们显式要求编译器生成=default的移动操作，且编译器不能移动所有成员，则编译器会将移动操作定义为删除的函数
    - 移动操作，拷贝操作，删除操作之间的关系
        - 定义了一个以上的移动操作，则默认的拷贝操作会被删除
        - 定义了一个以上的拷贝操作，默认不会定义移动操作
        - 定义了一个移动构造函数或者移动赋值运算符的类必须定义自己的拷贝操作。
- 在移动和拷贝同时出现在一个类的时候
    - 移动右值，拷贝左值
    - 没有移动构造函数的情况下，右值也被拷贝
        - 即使是调用std：：move也是如此
    - 使用拷贝构造函数代替移动构造函数一定是安全的，但是对于部分情况拷贝构造函数会损失性能
    - 如果一个类有一个可用的拷贝构造函数而没有移动构造函数，则其对象是通过拷贝构造函数来移动的。拷贝赋值运算符和移动赋值运算符情况类似
    - 右值：x&&，左值const x&；
    - 引用限定符
        - 有时候我们希望强制左侧运算对象（this指向的对象）是一个左值
        - 可以通过增加引用限定符，方法与使用const成员函数相同，&代表只可调用左值，&&代表只可调用右值，引用限定符必须跟在const限定符之后

```
class Foo {
public:
    Foo &operator=(const Foo&) &; // 只能向可修改的左值赋值
    // Foo 的其他参数
};
Foo &Foo::operator=(const Foo &rhs) &
{
    // 执行将 rhs 赋予本对象所需的工作
    return *this;
}
```

```
class Foo {
public:
Foo someMem() & const;        // 错误：const 限定符必须在前
Foo anotherMem() const &;     // 正确：const 限定符在前
};
```

        - 和成员函数可以根据是否右const来区分重载版本，引用限定符也可以区分重载版本，而且能和const一起使用
        - 对于有两个或者两个以上的具有相同名字的和相同参数的列表，引用限定要么都不

加，要么都加上

- 三五法则
    - 所有五个拷贝控制成员应该看作一个整体：一般来说如果一个类定义了任意一个拷贝操作，他就应该定义五个所有操作